CSE 332

JUNE 26TH – ANALYSIS OF THE HEAP

ASSORTED MINUTIAE

- Some problems with EX02 and EX03

ASSORTED MINUTIAE

- **Some problems with EX02 and EX03**
  - Deadline will be extended to Friday

ASSORTED MINUTIAE

- **Some problems with EX02 and EX03**
  - Deadline will be extended to Friday
  - If solutions aren't resolved by tomorrow, it will be converted to a pdf submission on canvas.

ASSORTED MINUTIAE

- **Some problems with EX02 and EX03**
  - Deadline will be extended to Friday
  - If solutions aren't resolved by tomorrow, it will be converted to a pdf submission on canvas.
- **P1 should be out and you should have begun work**

ASSORTED MINUTIAE

- **Some problems with EX02 and EX03**
  - Deadline will be extended to Friday
  - If solutions aren't resolved by tomorrow, it will be converted to a pdf submission on canvas.
- **P1 should be out and you should have begun work**
  - Checkpoint moved to Friday

TODAY'S LECTURE

- bigO and analysis

TODAY'S LECTURE

- bigO and analysis
- Analyzing the heap

TODAY'S LECTURE

- bigO and analysis
- Analyzing the heap
- Floyd's method

REVIEW FROM LAST WEEK

- **Heap implementation**

REVIEW FROM LAST WEEK

- **Heap implementation**
  - Complete

REVIEW FROM LAST WEEK

- **Heap implementation**
  - Complete
  - Heap property

REVIEW FROM LAST WEEK

- **Heap implementation**
  - Complete
  - Heap property
  - Array implementation (0 or 1 indexing)

REVIEW FROM LAST WEEK

- **Heap implementation**
  - Complete
  - Heap property
  - Array implementation (0 or 1 indexing)

REVIEW FROM LAST WEEK

- **Heap implementation**
  - Complete
  - Heap property
  - Array implementation (0 or 1 indexing)
  - Percolate up and percolate down

REVIEW FROM LAST WEEK

- **Heap implementation**
  - Complete
  - Heap property
  - Array implementation (0 or 1 indexing)
  - Percolate up and percolate down
  - *d*-heaps

REVIEW FROM LAST WEEK

- Algorithm analysis

REVIEW FROM LAST WEEK

- **Algorithm analysis**
  - Counting operations strictly is unreliable

REVIEW FROM LAST WEEK

- **Algorithm analysis**
  - Counting operations strictly is unreliable
  - Want some way for us to compare functions

REVIEW FROM LAST WEEK

- **Algorithm analysis**
  - Counting operations strictly is unreliable
  - Want some way for us to compare functions
  - bigO – asymptotic runtime bounds

REVIEW FROM LAST WEEK

- **Algorithm analysis**
  - Counting operations strictly is unreliable
  - Want some way for us to compare functions
  - bigO – asymptotic runtime bounds
  - `f(n) = O(g(n))` if there exists some c and n_0 such that `f(n) < c*g(n)` for some `c > 0` and all `n >` n_0

BIG-O NOTATION

- Big-O is for upper bounds.

BIG-O NOTATION

- Big-O is for upper bounds.
- It's equivalent for lower bounds is big Omega

BIG-O NOTATION

- Big-O is for upper bounds.
- It’s equivalent for lower bounds is big Omega

Formally, a function `f(n)` is Ω`(g(n))` if there exists a `c` and $n_0 > 0$ such that:

- For all $n \geq n_0$, `f(n) > c*g(n)`

BIG-O NOTATION

- Big-O is for upper bounds.
- It's equivalent for lower bounds is big Omega

Formally, a function `f(n)` is Ω`(g(n))` if there exists a `c` and $n_0 > 0$ such that:

- For all $n \geq n_0$, `f(n) > c*g(n)`

- If a function `f(n)` is in `O(g(n))` and Ω`(g(n))`

BIG-O NOTATION

- If a function `f(n)` is in `O(g(n))` and `Ω(g(n))`, then g(n) is a tight bound on f(n), we call this big theta.

BIG-O NOTATION

- If a function `f(n)` is in O`(g(n))` and Ω`(g(n))`, then g(n) is a tight bound on f(n), we call this big theta.
- Formally, if `f(n)` is in O`(g(n))` and Ω`(g(n))`, then `f(n)` is in θ`(g(n))`
- Note that the two will have different c and n_0

BIG O NOTATION

- **What does this help us with?**
  - Sort algorithms into families

BIG O NOTATION

- **What does this help us with?**
  - Sort algorithms into families
    - $O(1)$: constant
    - $O(\log n)$: logarithmic
    - $O(n)$: linear
    - $O(n^2)$: quadratic
    - $O(n^k)$: polynomial
    - $O(k^n)$: exponential

BIG O NOTATION

- **What does this help us with?**

BIG O NOTATION

- **What does this help us with?**
  - The constant multiple c lets us organize similar algorithms together.
  - Remember that $\log_a k$ and $\log_b k$ differ by a constant factor?

BIG O NOTATION

- **What does this help us with?**
  - The constant multiple c lets us organize similar algorithms together.
  - Remember that $\log_a k$ and $\log_b k$ differ by a constant factor?
  - That makes all logs in the same family

CORRECTNESS ANALYSIS

- **How do we show an algorithm is correct?**

CORRECTNESS ANALYSIS

- **How do we show an algorithm is correct?**
  - Need to look at the approach

BINARY SEARCH (AGAIN)

```
public int binarySearch(int[] data, int toFind){
int low = 0; int high = data.length-1;
while(low <= high){
      int mid = (low+high)/2;
      if(toFind>mid) low = mid+1; continue;
      else if(toFind<mid) high = mid-1; continue;
      else return mid;
}
return -1;
}
```

BINARY SEARCH CORRECTNESS

- Prove binary search returns the correct answer

BINARY SEARCH CORRECTNESS

- **Prove binary search returns the correct answer**
  - Need property of sortedness

BINARY SEARCH CORRECTNESS

- **Prove binary search returns the correct answer**
  - Need property of sortedness
  - For all pairs i,j in the array:
    - If `A[i]` $\leq$ `A[j]`, then `i` $\leq$ `j`

BINARY SEARCH CORRECTNESS

- **Prove binary search returns the correct answer**
  - Need property of sortedness
  - For all pairs i,j in the array:
    - If `A[i]` $\leq$ `A[j]`, then `i` $\leq$ `j`
  - Binary search always chooses the correct side

BINARY SEARCH CORRECTNESS

- **Prove binary search returns the correct answer**
  - Need property of sortedness
  - For all pairs i,j in the array:
    - If `A[i]` $\leq$ `A[j]`, then `i` $\leq$ `j`
  - Binary search always chooses the correct side
  - End case: low = high

ANALYSIS

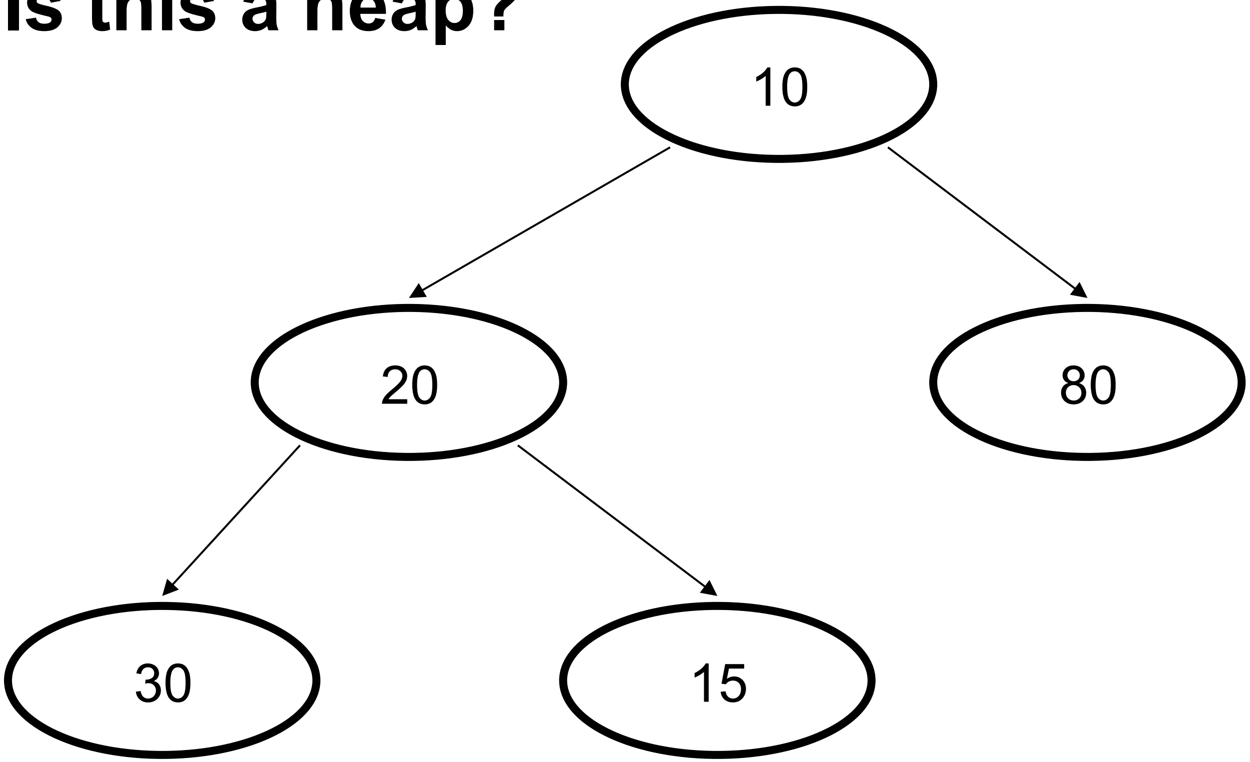
- **Let’s use these analytical approaches to solve some things about heap functions**

ANALYSIS

- Let’s use these analytical approaches to solve some things about heap functions
- First, let’s do a quick review of heap properties

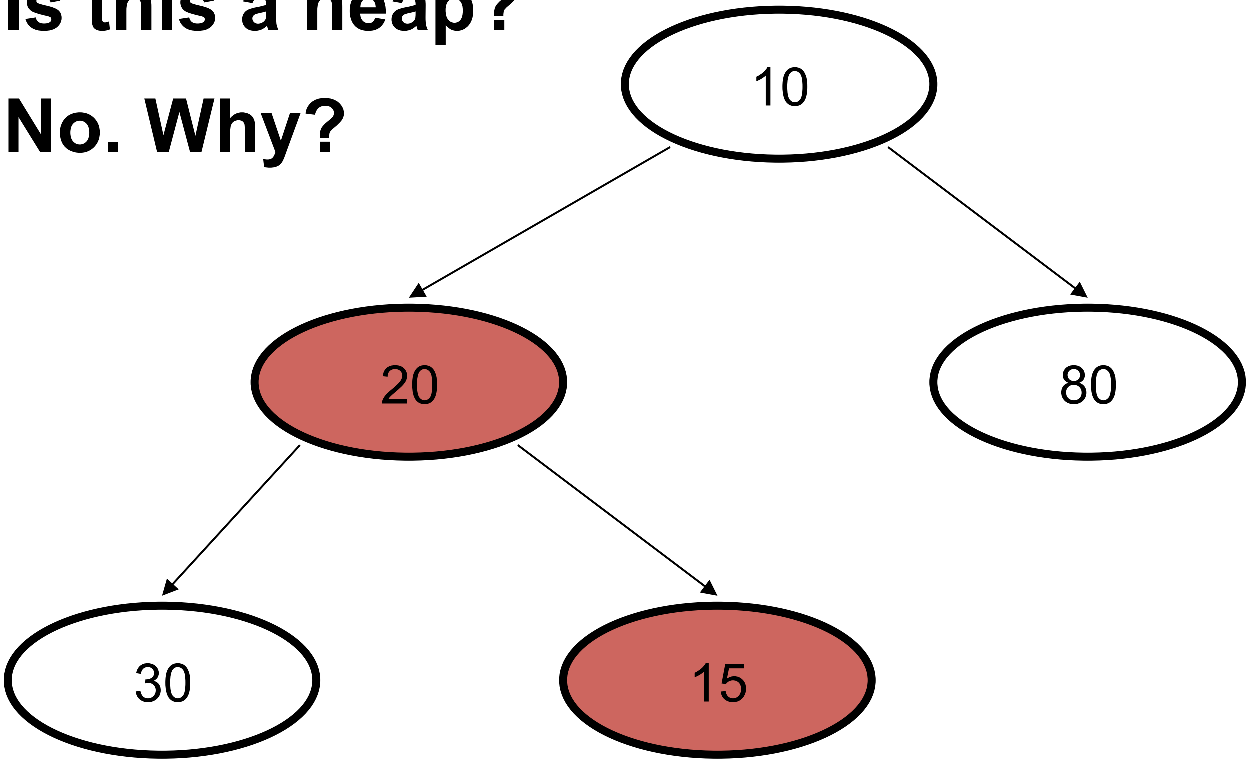
REVIEW

- **Is this a heap?**



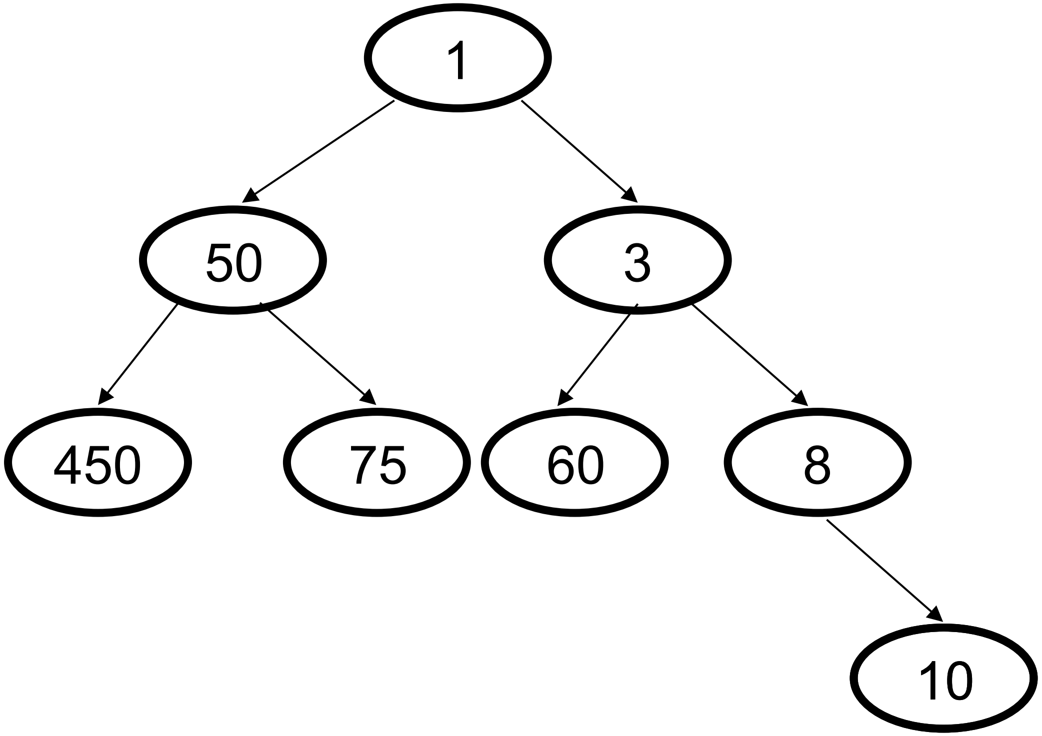
REVIEW

- **Is this a heap?**
- **No. Why?**



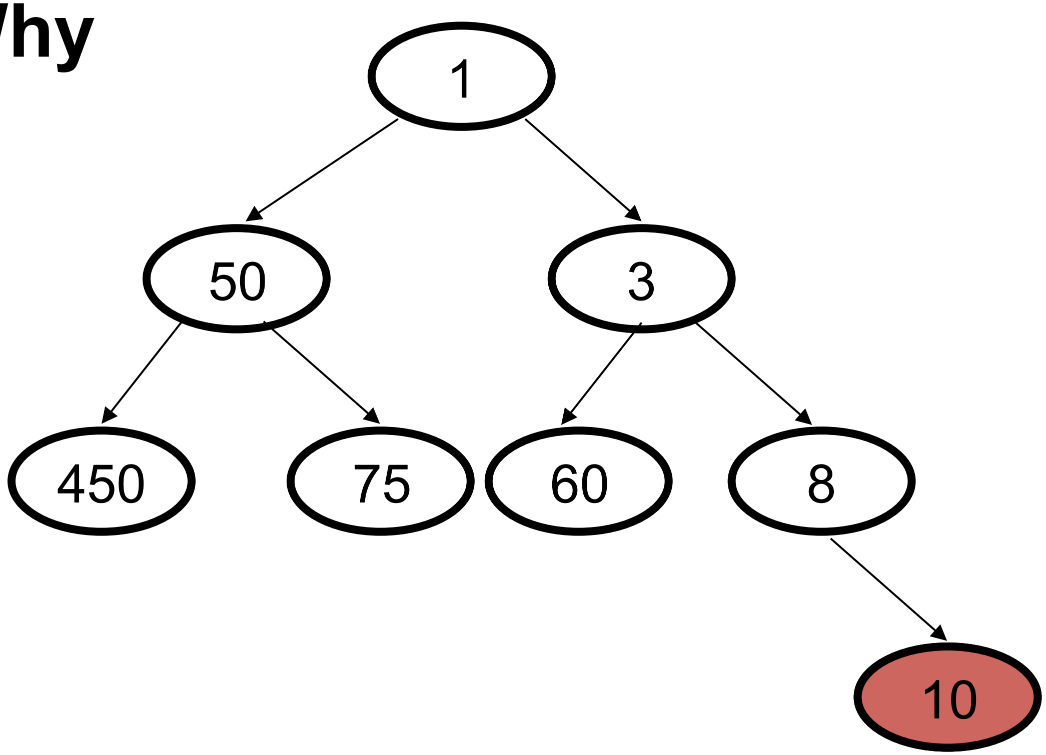
REVIEW

- **Is this a heap?**



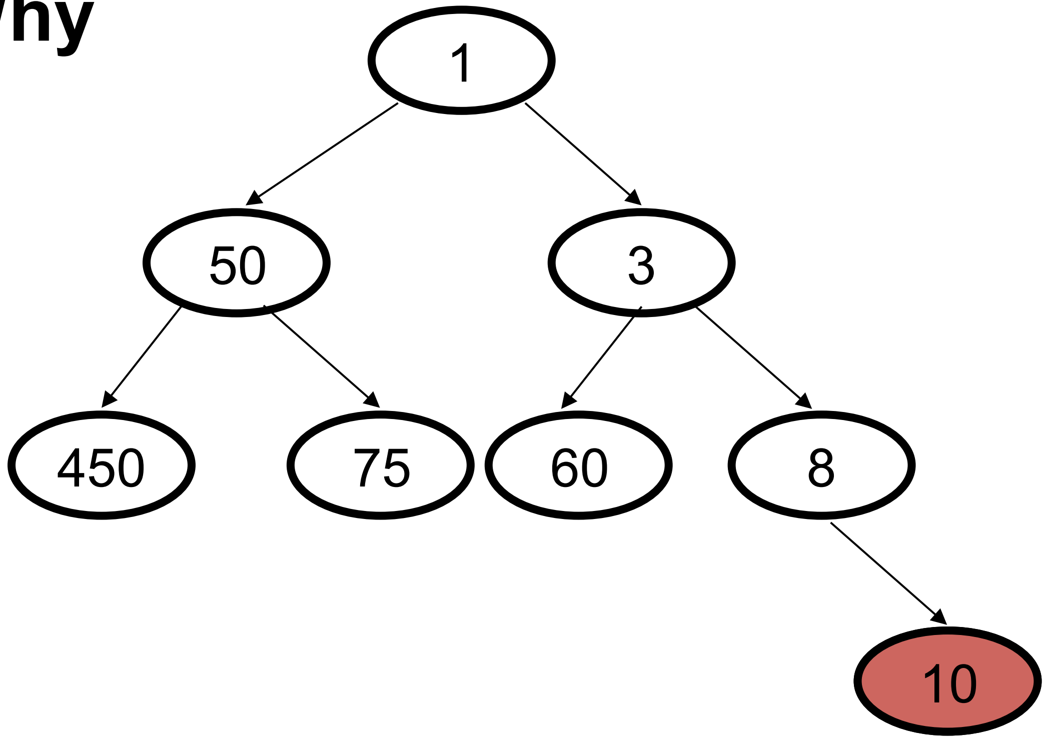
REVIEW

- **Is this a heap?**
- **No. Why**



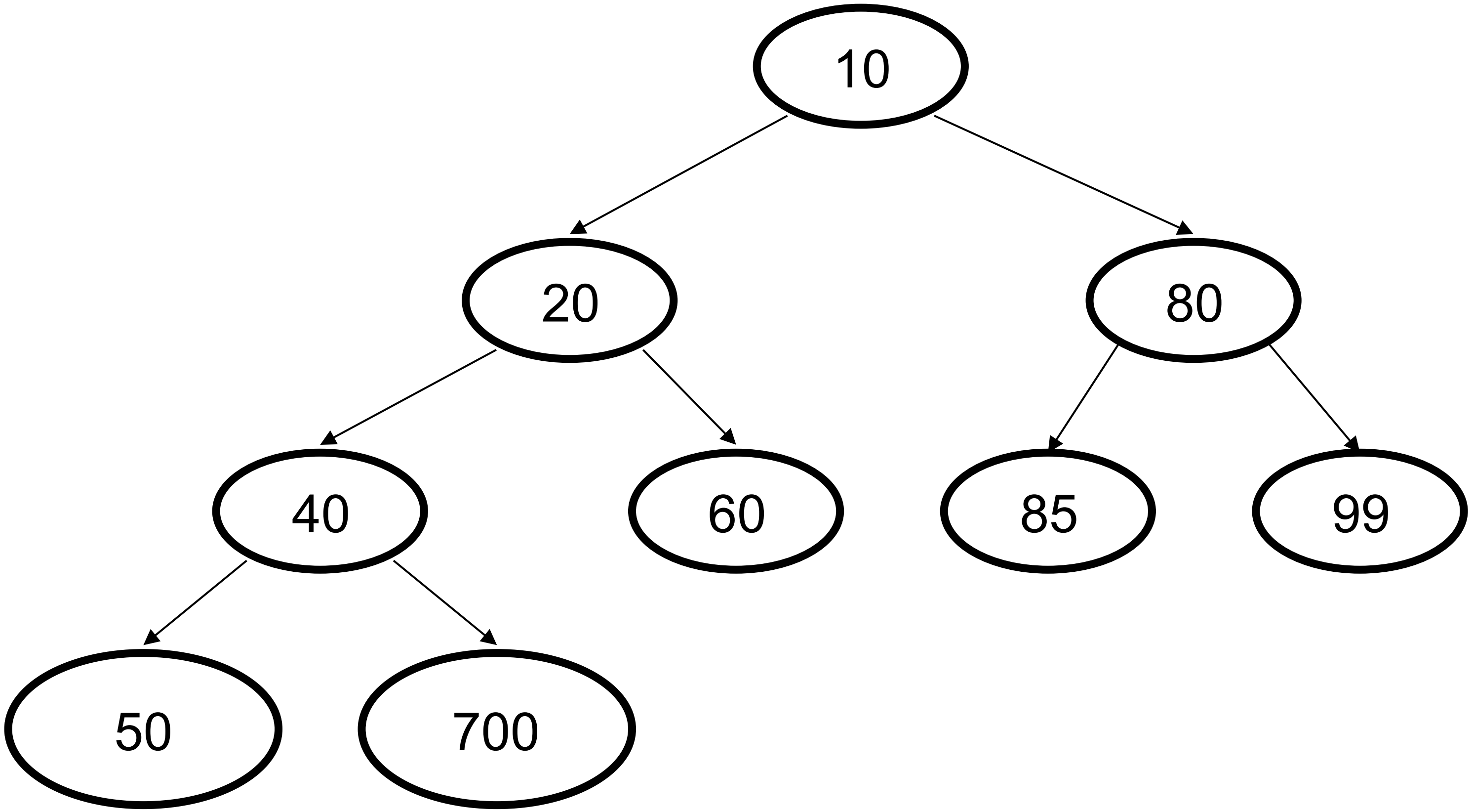
REVIEW

- Is this a heap?
- No. Why



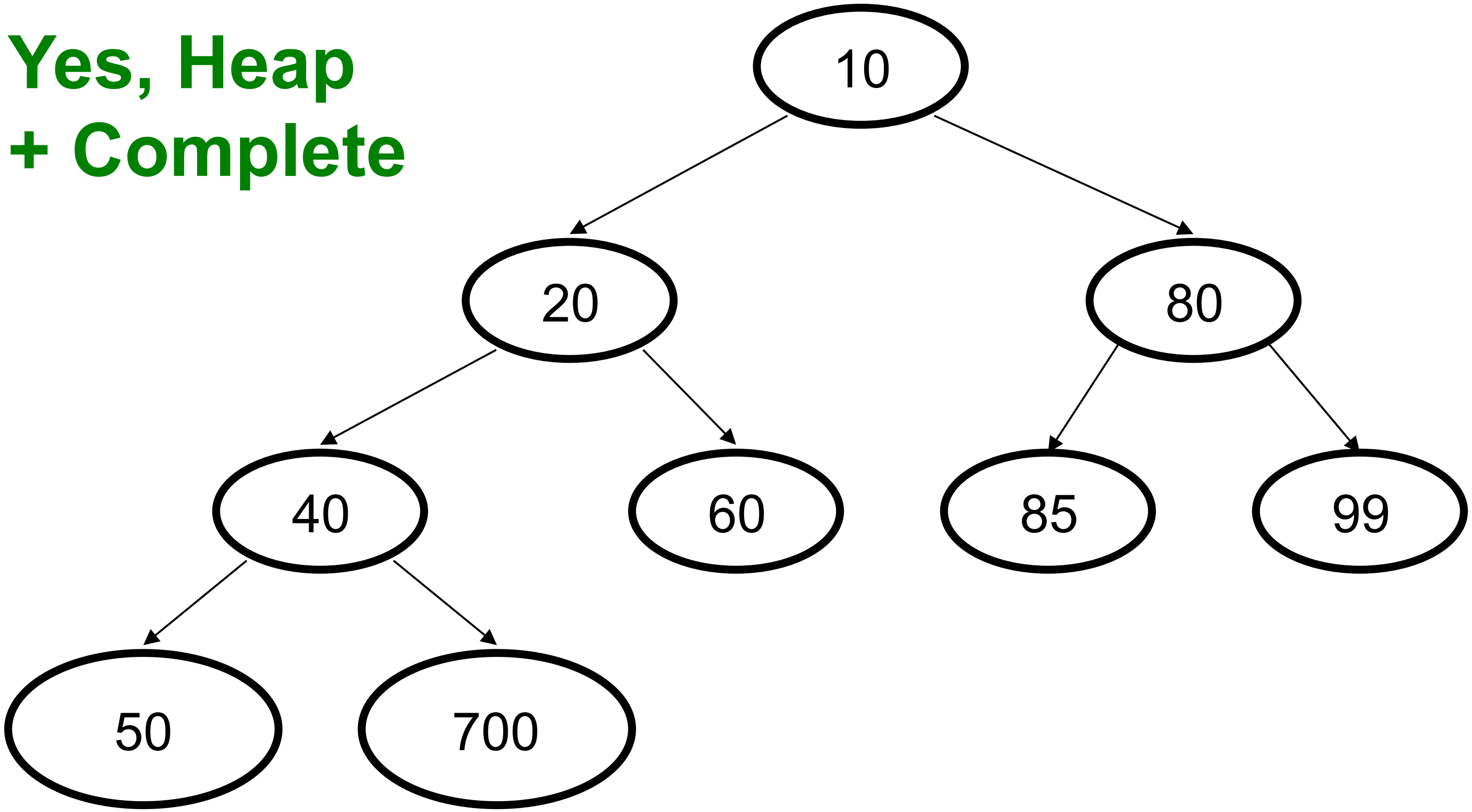
REVIEW

- **Is this a heap?**



REVIEW

- **Is this a heap?**
- **Yes, Heap + Complete**

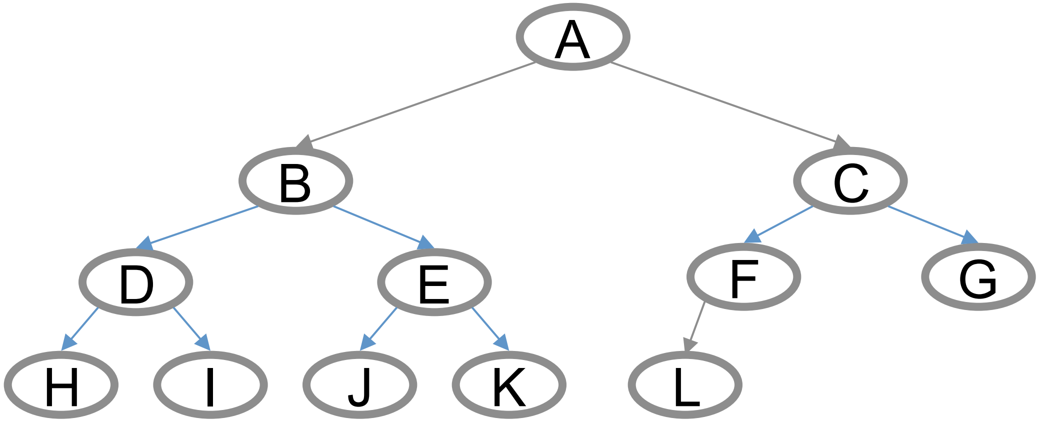


REVIEW

- **Heaps**
  - Properties
    - Completeness
    - Heap property
  - Implementation
    - Array (0 v 1 indexing)

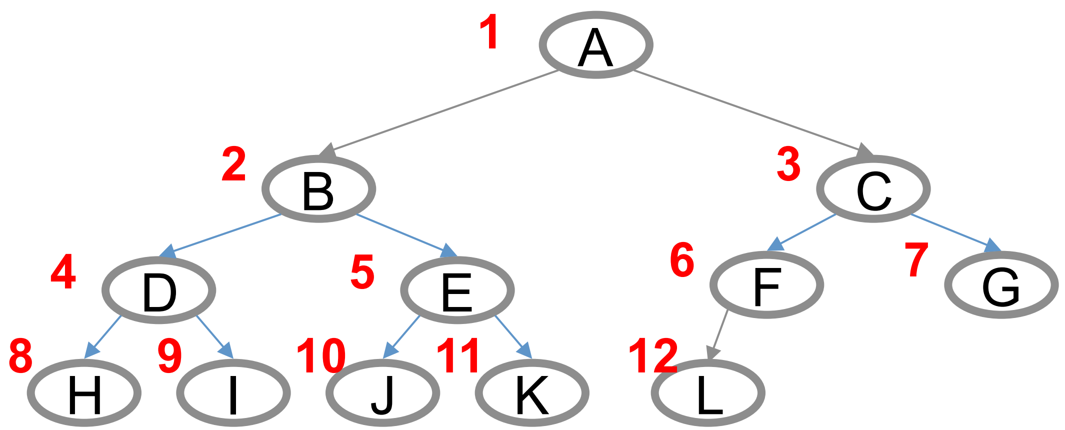
REVIEW

- **Array property**



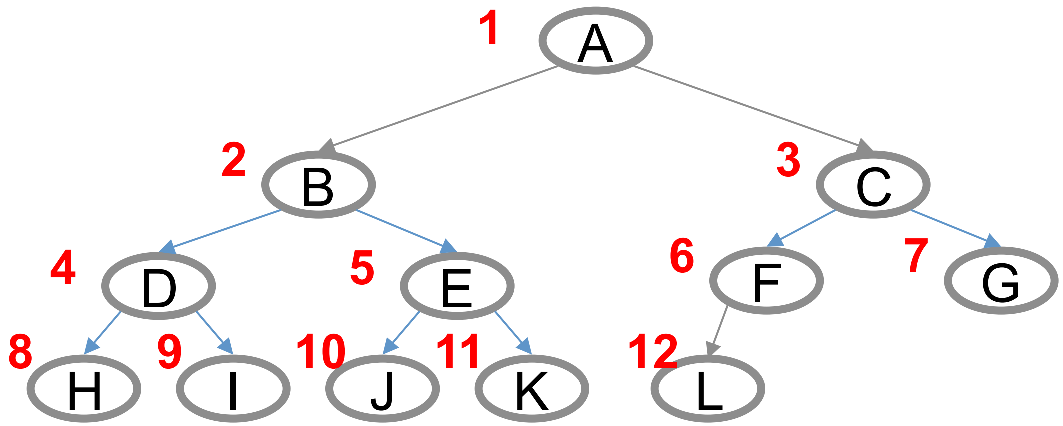
REVIEW

- **Array property**



REVIEW

- **Array property**



	A	**B**	**C**	**D**	**E**	**F**	**G**	**H**	**I**	**J**	**K**	**L**	
0	1	2	3	4	5	6	7	8	9	10	11	12	13

REVIEW

- **With 0 indexing:**
  - Parent:
  - Left-child:
  - Right-child:

REVIEW

- **With 0 indexing:**
  - Parent: (i-1)/2
  - Left-child:
  - Right-child:

REVIEW

- **With 0 indexing:**
  - Parent: (i-1)/2
  - Left-child: 2i+1
  - Right-child:

REVIEW

- **With 0 indexing:**
  - Parent: (i-1)/2
  - Left-child: 2i+1
  - Right-child: 2i+2

REVIEW

- **With 1 indexing:**
  - Parent:
  - Left-child:
  - Right-child:

REVIEW

- **With 1 indexing:**
  - Parent: i/2
  - Left-child:
  - Right-child:

REVIEW

- **With 1 indexing:**
  - Parent: i/2
  - Left-child: 2i
  - Right-child:

REVIEW

- **With 1 indexing:**
  - Parent: i/2
  - Left-child: 2i
  - Right-child: 2i+1

REVIEW

- **What about for a *d*-heap?**

REVIEW

- **What about for a *d*-heap?**
- **Arithmetic changes slightly, but it is still doable**

HEAPS

- **Operations**
  - Insert: adds a data, priority pair into the heap

HEAPS

- **Operations**
  - Insert: adds a data, priority pair into the heap
  - deleteMin: returns and removes the item of smallest priority from the heap

HEAPS

- **Operations**
  - Insert: adds a data, priority pair into the heap
  - deleteMin: returns and removes the item of smallest priority from the heap
  - changePriority: changes the priority of a particular item in the heap

HEAPS

- **Operations**
  - Insert: adds a data, priority pair into the heap
  - deleteMin: returns and removes the item of smallest priority from the heap
  - changePriority: changes the priority of a particular item in the heap

HEAPS

- **Operations**
  - Insert: adds a data, priority pair into the heap
  - deleteMin: returns and removes the item of smallest priority from the heap
  - changePriority: changes the priority of a particular item in the heap
- **What are the (worst-case) runtimes for these operations?**

HEAPS

- **Insert:**
  - Add the element at the bottom of the tree

HEAPS

- **Insert:**
  - Add the element at the bottom of the tree
  - “Percolate up” that element to its correct place

HEAPS

- **Insert:**
  - Add the element at the bottom of the tree
  - "Percolate up" that element to its correct place
- **Adding to the end of a tree?**

HEAPS

- **Insert:**
  - Add the element at the bottom of the tree
  - “Percolate up” that element to its correct place
- **Adding to the end of a tree?** `O(1)`

HEAPS

- **Insert:**
  - Add the element at the bottom of the tree
  - “Percolate up” that element to its correct place
- **Adding to the end of a tree? `O(1)`**

HEAPS

- **Insert:**
  - Add the element at the bottom of the tree
  - "Percolate up" that element to its correct place
- **Adding to the end of a tree?** **`O(1)`**
- **Percolating up?**

HEAPS

- **Insert:**
  - Add the element at the bottom of the tree
  - “Percolate up” that element to its correct place
- **Adding to the end of a tree? `O(1)`**
- **Percolating up? `O(height)`**

HEAPS

- **Insert:**
  - Add the element at the bottom of the tree
  - "Percolate up" that element to its correct place
- **Adding to the end of a tree? `O(1)`**
- **Percolating up? `O(height)`**
  - What is the height of a heap?

HEAPS

- **Insert:**
  - Add the element at the bottom of the tree
  - "Percolate up" that element to its correct place
- **Adding to the end of a tree? `O(1)`**
- **Percolating up? `O(height)`**
  - What is the height of a heap? $\log_2 n$

HEAPS

- **Insert:**
  - Add the element at the bottom of the tree
  - “Percolate up” that element to its correct place
- **Adding to the end of a tree? `O(1)`**
- **Percolating up?** ~~**`O(height)`**~~ **`O(log n)`**
  - What is the height of a heap? $\log_2 n$

HEAPS

- deleteMin:

HEAPS

- **deleteMin:**
  - Move the last element up to the top of the tree
  - Percolate that element down
  - Return the original root of the tree.

HEAPS

- **deleteMin:**
  - Move the last element up to the top of the tree
  - Percolate that element down
  - Return the original root of the tree.
- **Copying element?**

HEAPS

- **deleteMin:**
  - Move the last element up to the top of the tree
  - Percolate that element down
  - Return the original root of the tree.
- **Copying element? O(1)**

HEAPS

- **deleteMin:**
  - Move the last element up to the top of the tree
  - Percolate that element down
  - Return the original root of the tree.
- **Copying element? O(1)**
- **Percolating down?**

HEAPS

- **deleteMin:**
  - Move the last element up to the top of the tree
  - Percolate that element down
  - Return the original root of the tree.
- **Copying element? O(1)**
- **Percolating down? O(log n)**

HEAPS

- **deleteMin:**
  - Move the last element up to the top of the tree
  - Percolate that element down
  - Return the original root of the tree.
- **Copying element? O(1)**
- **Percolating down? O(log n)**
- **Returning element?**

HEAPS

- **deleteMin:**
  - Move the last element up to the top of the tree
  - Percolate that element down
  - Return the original root of the tree.
- **Copying element? O(1)**
- **Percolating down? O(log n)**
- **Returning element? O(1)**

HEAPS

- changePriority:

HEAPS

- **changePriority:**
  - Find the element
  - Percolate up/down

HEAPS

- **changePriority:**
  - Find the element
  - Percolate up/down
- **Finding in a heap?**

HEAPS

- **changePriority:**
  - Find the element
  - Percolate up/down
- **Finding in a heap? O(n)**

HEAPS

- **changePriority:**
  - Find the element
  - Percolate up/down
- **Finding in a heap? O(n) Why?**

HEAPS

- **changePriority:**
  - Find the element
  - Percolate up/down
- **Finding in a heap? O(n) Why?**
  - Heap property does not give us the divide and conquer benefit

HEAPS

- **changePriority:**
  - Find the element
  - Percolate up/down
- **Finding in a heap? O(n) Why?**
  - Heap property does not give us the divide and conquer benefit
- **Percolate up/down?**

HEAPS

- **changePriority:**
  - Find the element
  - Percolate up/down
- **Finding in a heap? O(n) Why?**
  - Heap property does not give us the divide and conquer benefit
- **Percolate up/down? O(log n)**

HEAPS

- **changePriority:**
  - Find the element
  - Percolate up/down
- **Finding in a heap? O(n) Why?**
  - Heap property does not give us the divide and conquer benefit
- **Percolate up/down? O(log n)**
- **On average, is it faster to percolate up or down?**

ANALYSIS

- Let’s find an interesting algorithm to analyze

ANALYSIS

- Let’s find an interesting algorithm to analyze
- Given an array of length n, how do we make that array into a heap?

ANALYSIS

- **Let's find an interesting algorithm to analyze**
- **Given an array of length n, how do we make that array into a heap?**
- **Naïve approach?**
  - Make a new heap and add each element of the array into the heap

ANALYSIS

- **Let's find an interesting algorithm to analyze**
- **Given an array of length n, how do we make that array into a heap?**
- **Naïve approach?**
  - Make a new heap and add each element of the array into the heap
  - How long to finish?

FUN FACTS!

- **Is it really O(n log n)?**

FUN FACTS!

- **Is it really O(n log n)?**
  - Early insertions are into empty trees

FUN FACTS!

- **Is it really O(n log n)?**
  - Early insertions are into empty trees **O(1)**!

FUN FACTS!

- **Is it really O(n log n)?**
  - Early insertions are into empty trees **O(1)**!
  - Consider a simpler example, adding to the end of a linked list.

FUN FACTS!

- **Is it really O(n log n)?**
  - Early insertions are into empty trees **O(1)**!
  - Consider a simpler example, creating a sorted linked list.
  - Adding at the end of a linked list with k items takes O(k) operations.

FUN FACTS!

- **Is it really O(n log n)?**
  - Early insertions are into empty trees **O(1)**!
  - Consider a simpler example, creating a sorted linked list.
  - Adding at the end of a linked list with k items takes O(k) operations.

FUN FACTS!

- **Is it really O(n log n)?**
  - Early insertions are into empty trees **O(1)**!
  - Consider a simpler example, creating a sorted linked list.
  - Adding at the end of a linked list with k items takes O(k) operations.

  1+2+3+4+5…

FUN FACTS!

- **Is it really O(n log n)?**
  - Early insertions are into empty trees **O(1)**!
  - Consider a simpler example, creating a sorted linked list.
  - Adding at the end of a linked list with k items takes O(k) operations.

1+2+3+4+5…

FUN FACTS!

- **Is it really O(n log n)?**
  - Early insertions are into empty trees **O(1)**!
  - Consider a simpler example, creating a sorted linked list.
  - Adding at the end of a linked list with k items takes O(k) operations.

1+2+3+4+5…

What is this summation?

FUN FACTS!

$$\sum_{k=1}^{n} k = \frac{1}{2} n (n + 1)$$

FUN FACTS!

$$\sum_{k=1}^{n} k = \frac{1}{2} n (n+1)$$

- **What does this mean?**

FUN FACTS!

$$\sum_{k=1}^{n} k = \frac{1}{2} n(n+1)$$

- What does this mean?
- Summing k from 1 to n is still $O(n^2)$

FUN FACTS!

$$\sum_{k=1}^{n} k = \frac{1}{2} n (n+1)$$

- **What does this mean?**
- **Summing k from 1 to n is still `O(n`^{2}`)`**
- **Similarly, summing `log(k)` from 1 to n is `O(n log n)`**

ANALYSIS

- **Naïve approach:**
  - Must add n items

ANALYSIS

- **Naïve approach:**
  - Must add n items
  - Each add takes how long?

ANALYSIS

- **Naïve approach:**
  - Must add n items
  - Each add takes how long? `log(n)`

ANALYSIS

- **Naïve approach:**
  - Must add n items
  - Each add takes how long? `log(n)`
  - Whole operation is `O(log(n))`

ANALYSIS

- **Naïve approach:**
  - Must add n items
  - Each add takes how long? `log(n)`
  - Whole operation is `O(log(n))`
  - Can we do better?

ANALYSIS

- **Naïve approach:**
  - Must add n items
  - Each add takes how long? `log(n)`
  - Whole operation is `O(log(n))`
  - Can we do better?
    - What is better?

ANALYSIS

- **Naïve approach:**
  - Must add n items
  - Each add takes how long? `log(n)`
  - Whole operation is `O(log(n))`
  - Can we do better?
    - What is better? O(n)

HEAPS

- **Facts of binary trees**

HEAPS

- **Facts of binary trees**
  - Increasing the height by one doubles the number of possible nodes

HEAPS

- **Facts of binary trees**
  - Increasing the height by one doubles the number of possible nodes
  - Therefore, a complete binary tree has half of its nodes in the leaves

HEAPS

- **Facts of binary trees**
  - Increasing the height by one doubles the number of possible nodes
  - Therefore, a complete binary tree has half of its nodes in the leaves
  - A new piece of data is much more likely to have to percolate down to the bottom than be the smallest item in the heap

BUILDHEAP

- **So a naïve buildheap takes O(n log n)**

BUILDHEAP

- **So a naïve buildheap takes O(n log n)**
  - Why implement at all?

BUILDHEAP

- **So a naïve buildheap takes O(n log n)**
  - Why implement at all?
  - If we can get it `O(n)`!

FLOYD'S METHOD

- **Traverse the tree from bottom to top**
  - Reverse order in the array

FLOYD'S METHOD

- **Traverse the tree from bottom to top**
  - Reverse order in the array
- **Start with the last node that has children.**
  - How to find?

FLOYD'S METHOD

- **Traverse the tree from bottom to top**
  - Reverse order in the array
- **Start with the last node that has children.**
  - How to find? `Size / 2`

FLOYD'S METHOD

- **Traverse the tree from bottom to top**
  - Reverse order in the array
- **Start with the last node that has children.**
  - How to find? `Size / 2`
- **Percolate down each node as necessary**

FLOYD'S METHOD

- **Traverse the tree from bottom to top**
  - Reverse order in the array
- **Start with the last node that has children.**
  - How to find? `Size / 2`
- **Percolate down each node as necessary**
  - Wait! Percolate down is O(log n)!
  - This is an O(n log n) approach!

FLOYD'S METHOD

- It is O(n log n), because big O is an upper bound, but there is a tighter analysis possible!

FLOYD'S METHOD

- It is O(n log n), because big O is an upper bound, but there is a tighter analysis possible!
- How far does each node travel (at worst)

FLOYD'S METHOD

- **It is O(n log n), because big O is an upper bound, but there is a tighter analysis possible!**
- **How far does each node travel (at worst)**
  - Leaves don't move at all: Height = 0

FLOYD'S METHOD

- **It is O(n log n), because big O is an upper bound, but there is a tighter analysis possible!**
- **How far does each node travel (at worst)**
  - Leaves don't move at all: Height = 0
    - This is half the nodes in the tree

FLOYD'S METHOD

- **It is O(n log n), because big O is an upper bound, but there is a tighter analysis possible!**
- **How far does each node travel (at worst)**
  - 1/2 of the nodes don't move:
    - These are leaves – Height = 0
  - 1/4 can move at most one
  - 1/8 can move at most two

FLOYD'S METHOD

- **It is O(n log n), because big O is an upper bound, but there is a tighter analysis possible!**
- **How far does each node travel (at worst)**
  - 1/2 of the nodes don't move:
    - These are leaves – Height = 0
  - 1/4 can move at most one
  - 1/8 can move at most two …

FLOYD'S METHOD

$$\sum_{i=0}^{n} \frac{i}{2^{i+1}} = 2^{-n-1}\left(-n + 2^{n+1} - 2\right)$$

- Thanks Wolfram Alpha!

FLOYD'S METHOD

$$\sum_{i=0}^{n} \frac{i}{2^{i+1}} = 2^{-n-1}\left(-n + 2^{n+1} - 2\right)$$

- Thanks Wolfram Alpha!
- Does this look like an easier summation?

FLOYD'S METHOD

$$\sum_{i=0}^{\infty} \frac{1}{2^{i+1}} = 1$$

FLOYD'S METHOD

$$\sum_{i=0}^{\infty} \frac{1}{2^{i+1}} = 1$$

- **This is a must know summation!**

FLOYD'S METHOD

$$\sum_{i=0}^{\infty} \frac{1}{2^{i+1}} = 1$$

- This is a must know summation!
- 1/2 + 1/4 + 1/8 + … = 1

FLOYD'S METHOD

$$\sum_{i=0}^{\infty} \frac{1}{2^{i+1}} = 1$$

- This is a must know summation!
- 1/2 + 1/4 + 1/8 + … = 1
- How do we use this to prove our complicated summation?

FLOYD'S METHOD

$1/2 + 1/4 + 1/8 \ldots \quad \ldots + 1/2^n = 1$

FLOYD'S METHOD

$$1/2 + 1/4 + 1/8 \ldots \quad \ldots + 1/2^n = 1$$

$$1/4 + 1/8 \ldots \quad \ldots + 1/2^n = 1/2$$

$$1/8 \ldots \quad \ldots + 1/2^n = 1/4$$

FLOYD'S METHOD

$$\begin{aligned} 1/2 + 1/4 + 1/8 \ldots \quad \ldots + 1/2^n &= 1 \\ 1/4 + 1/8 \ldots \quad \ldots + 1/2^n &= 1/2 \\ 1/8 \ldots \quad \ldots + 1/2^n &= 1/4 \end{aligned}$$

- **Vertical columns sum to:**
  `i/2^i`, which is what we want

- **What is the right summation?**
  - Our original summation plus 1

FLOYD'S METHOD

$$\sum_{i=1}^{\infty} \frac{i}{2^i} = 2$$

FLOYD'S METHOD

$$\sum_{i=1}^{\infty} \frac{i}{2^i} = 2$$

- **This means that the number of swaps we perform in Floyd's method is 2 times the size… So Floyd's method is `O(n)`**

NEXT LECTURE

- Back to analysis

NEXT LECTURE

- Back to analysis
- Recurrences and analyzing recursive functions